

PDP-1 COMPUTER
ELECTRICAL ENGINEERING DEPARTMENT
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
CAMBRIDGE, MASSACHUSETTS
02139

PDP-35

INSTRUCTION MANUAL

PART 5A -- BASIC SYSTEM CALLS

1 February 1975

Table of Contents

5A	0	Introduction and Background
	1	The W register
	2	Low Priority Mode
	3	Real Time Clock
	4	Virtual Memory Assignment
	5	Illegal Instruction Return/ Illegal Memory Reference Return
	6	Concept of Capability; Mta's and Ivk's Differentiated
	7	General Capability Manipulation
	8	PRL Mode
	9	External Equipment
	10.0	Mta 30X description
	10.1	Drum Fields
	10.2	Queues
	10.3	Directories
	10.4	Files
	10.5	Using I/O Devices
	10.5.1	Microtapes
	10.5.2	Button Consoles
	10.5.3	Temporary Clock
	10.5.4	Paper Tape Reader and Punch
	10.5.5	Typewriters
	10.5.6	Hardware I/O Devices
	11	Disowned Objects
	12	Comments, Randomness

5B Advanced System Features

	0	Advanced System Features
	0.1	Elaboration on Concept of Spheres
	0.2	Protection, Intersphere Communication
	1	Simple Sphere Control
	2	Attachments
	3.0	Entry Capabilities
	3.1	Master Entry ivks
	3.2	Examples of Use
	4.0	Traps, creation of EPC
	4.1	EPC Ivks
	4.2	Examples of entry trap
	5.0	Ownership, master/non-master spheres subjugation
	5.1	Fault traps
	5.2	Examples
	6.0	Breakpoints and ESI
	6.1	Examples
	7	Comments, Randomness

Summary of Mta's and Ivk's

(omit on first reading)

mta 0	Copy A into W.	5A.1
mta 1	Copy I into W.	5A.1
mta 2	Copy W into A.	5A.1
mta 3	Copy W into I.	5A.1
mta 4	= hlt. Cause mta 4 trap.	5B.5.1
mta 5	Cause mta 5 trap.	5B.5.1
mta 6	Cause mta 6 trap.	5B.5.1
mta 7	= dsm. Cause mta 7 trap.	5B.5.1
mta 100	Enter low priority mode.	5A.2
mta 103	Real Time Clock.	Instruction Memo part 3
mta 104	Read absolute drum fields 0-77	5A.10.1
	A(6-12) contains count/40	A(13) is write bit
	I(0-5) abs. field number	I(6-12) drum address
	W(3-17) core address	
mta 105	Read absolute drum fields 100-177	5A.10.1
mta 200	Read illegal instruction return into A.	5A.5
mta 201	Set illegal instruction return from A(3-17).	5A.5
mta 202	Read illegal memory reference return into A.	5A.5
mta 203	Set illegal memory reference return from A(3-17).	5A.5
mta 204	Delete capability given in A(12-17).	5A.7
mta 205	Detach core. If nonzero, A(3-5), else A(15-17).	5B.2
mta 206	Read memory bound and attachments into A.	5A.4
mta 207	Set memory bound. If nonzero, A(3-5), else A(15-17).	
	Skip.	5A.4

Note: For all mta 30X instructions, the low six bits of the AC, A(12-17), should contain the desired index of the created capability. If this field contains a zero, the first free capability index will be used. Skip.

mta 300 Assign drum field. If A(0) is zero, field assigned is read/write. If A(0) is one, absolute field specified in A(5-11) is assigned as read only.

Use:

A(13)	is write bit.	A(6-12)	is count/40.
I(6-17)	is drum address. Must be multiple of 40.		
W(3-17)	is core address.		

5A.10.1

mta 302

Create sphere. I(3-17) contains fault entry address.
5B.1

Use: A(14)=0, read/write sphere. 5B.1
A(6-12) holds count/40 A(13) write bit
I(3-17) sphere address. Multiple of 40.
W(3-17) core address.

AC Action

12 Suppress processing. 5B.1
32 Permit processing. 5B.1
52 Attach. I(15-17) attached as core I(3-5). 5B.2
72 Reverse attach. I(15-17) attached as core I(3-5).
112 Read process state. 5B.1
I process number
W(3-17) six word area to receive A,G,I,X,F,W.
Run indicator off. Skip.

132 Write process state. Similar to 112. 5B.1
152 Read breakpoint state. 5B.6
I(3-17) core address
Run indicator off. No skip.

172 Write breakpoint state. Similar to 152. 5B.6
412 Create process. Skip. Process no. returned in A.
Run indicator off. 5B.1

432 Delete process. Beware of process hoard. 5B.1
I(0-17) process number
Run indicator off. Skip.

452 Count processes. Answer in A. 5B.1
472 Set console assignment. 5B.1
A(0) zero, use ivk's console
A(0) one, use console 0 (on bay 11)

512 Set Trap Status. 5B.5
I(3) on to enable mta-trap
I(4) on to extend superiority
I(5) on to nullify this ivk in inferior

532 Read fault entry and superior. 5B.5
A contains entry address or is unchanged
I contains sphere number or 0

552 Subjugate. I(3-17) fault entry address. 5B.5
I(0-2) clear. Skip. Enters may occur.

572 Execute mta. A(0-8) contains mta code ≥ 200 5B.1
W contains AC to be used for executed mta
I contains I to be used for executed mta

612 Reverse share. 5B.1
I(6-11) is donor index.
I(12-17) is receiver index. If zero, first free.

632 Share. 5B.1
652 Reverse grant. 5B.1
672 Grant. 5B.1

mta 303 Create program queue. $-|I|$ is initial population.
 Negative numbers are taken as one's mode. Use $I \geq 0$.
 5A.10.2 and Instruction Memo part 4

Use:	Variant	Action
	0	Use variant specified by $A(13-14) + 1$.
	1	Enter queue.
	2	Release queue.
	3	Read queue population into A.

mta 304 Create directory. 5A.10.3

Use:	Variant	Action
	0	$A(8-11) + 1$ used instead
	1	Retrieve. $(dir, I(6-11)) \rightarrow (user, I(12-17))$
	2	Place. $(user, I(6-11)) \rightarrow (dir, I(12-17))$
	3	Remove. $(dir, A(12-17))$ goes away.
	7	Count capabilities in directory. Returned in A.
	14	Turn into read-only capability.
	15	Translate capability in $(dir, A(12-17))$. Returned in A.

mta 305 Create file. 5A.10.4

Use:	Variant	Action
	0	If $A(14)=1$ then $A(10-13) + 1$ is used for the variant. If $A(14)=0$ then read/write file is specified. $A(0-12)$ count. $A(13)$ write bit. $I(0-17)$ file address. (multiple of 40) $W(3-17)$ user core address. Current restrictions prevent writing across 400-word file boundary or user core boundary.
	1	Read $(length)/400$ into A.
	2	Set length from I to equal $(I) \times 400$ words.
	3	Convert to read-only file capability.

mta 306 Assign I/O device. ARG2 = A(6-11) CODE = A(0-5) 5A.10.
CODE=0 microtape unit given in ARG2. 5A.10.5.1
Use: A(0-2) zero. A(3-9) count/400.
A(10-11) zero. A(12) 0 translate, 1 no translation
A(13-17), 06 read, 26 write, 16 rew. wait, 36 rew
I contains tape address (multiple of 400)
W contains core address (multiple of 40)
CODE=1 buttons, console given in ARG2. 5A.10.5.2
Use: Hang until button status differs from A(0-17).
CODE=2 temporary clock. 5A.10.5.3
Use: 1760 decimal ticks per minute.
Hang for -A(0-17) ticks, unless C(A) > 777777
CODE=3 paper tape reader. ARG2, 0 alpha, 1 binary 5A.10.5.4
Use: data right justified in A. skip unless out of tape.
CODE=4 paper tape punch. 5A.10.5.4
Use: data taken from A(10-17).
CODE=5 typewriter, ARG2 specifies desired console. 5A.10.5.5
Use: 0 Use variant given by A(6-9) +1 .
1 A(12-17) typed out.
2 Type in to A(12-17).
3 I(12-17) typed out.
4 Type in to I(12-17).
5 Enable.
6 Disable.
11 Turn off enable/disable.
12 Convert to inferior.
13 A(11-17) typed out.
14 Type in to A(11-17).
15 I(11-17) typed out.
16 Type in to I(11-17).
CODE=6 Call button. ARG2 specifies console number.
Use: Hang until call button is pushed.
CODE=7 Hardware device. ARG2 specifies which device.

mta 307 Create entry capability. I(3-17) specifies entry address.
Use: ivk <master entry> 5B.3.1
A(12-17) gives new transmitted word
ivk <non-master entry> 5B.3.1
causes entry into sphere containing master entry

mta 400 Translate capability in A(12-17). Result in A. 5A.7
mta 401 Exchange capabilities A(6-11) and A(12-17). 5A.7
mta 402 Turn off PRL mode. 5A.8
mta 403 Turn on PRL mode. 5A.8
mta 404 Count capabilities. Result in A, 777777 if no C-list. 5A.7
mta 405 Duplicate capability in I(6-11) onto I(12-17),
first free. 5A.7
mta 406 Read process hoard into A. 5B.4
mta 407 Set process hoard from A. 5B.4

mta 500 Assign/deassign external equipment. 5A.9
 A(0-5), 0 assign external levels
 1 deassign external levels
 2 assign external register shared
 3 assign external register private
 4 deassign external register.
 External levels are:
 4 External clock
 7 Radio Astronomy antenna
 External levels are specified by A(i+10) for $1 \leq i \leq 7$.
 mta 501 Disown capability specified in I(6-11). Reference index
 returned in A. 5A.11
 mta 502 Claim capability. Disowned capability at reference index
 given in I(6-11) is claimed onto I(12-17), first free.

Preface

Instruction Memo part 5, composed of two parts, has been written to provide a technical description and a theoretical understanding of the PDP-1X system architecture. In addition to providing the basic reference material describing the available system calls, this memo attempts to provide the reader with some of the conceptual background needed to make intelligent use of these calls.

The material is arranged like a book - each section assumes at most the knowledge of previous sections. New users are urged to become familiar with the basic system features, as described in Instruction Memos 1 through 4, before plunging into the more advanced features. Such a policy will result in programs with fewer trivial bugs and in less aggravation for both you and the system hackers.

This Instruction Memo is concerned with the description of two classes of instructions: mta's and ivk's. In order to understand these instructions it is necessary to define certain frequently used terms. It is also desirable to have a certain amount of background knowledge of time sharing systems in general and to know the PDP-1 philosophy in particular. The following is intended as a concise general introduction, but does assume some familiarity with computer hardware and with common terminology of computer software.

The PDP-1, operating out of time sharing mode, is similar to an IBM 1130 or PDP-11 in its typical mode of operation: it has at most one user and all system resources are his to use or not use as he chooses. There is one central processing unit (CPU) which physically contains

one accumulator	{A or AC}	18 bits
one I register	{I or IO}	18 bits
one index register	{X}	18 bits

and a core-rename register (CR) 18 bits long not used out of time sharing. There is a fifteen bit program counter (PC) and a large number of individual "status" or "mode" bits such as the program flags, address mode bits, etc. Instructions to be executed by the CPU must be fetched directly from core memory, of which there physically exist five 4K-word 18 bit/word memories, numbered 0, 1, 2, 3 and 7.

One of the first, most obvious problems of operating this hardware in a time sharing mode is that there is physically only one CPU, whereas two or more users may wish to be running their programs. Solving the problem involves writing a program which can equitably schedule CPU time among the users who wish to run. When the scheduler decides the current user has run long enough and that someone else should now be allowed to run, it must save the contents of the hardware registers. When the descheduled user is next given time, if ever, his computation may be resumed after the registers are reloaded from the values saved. On the PDP-1, extra hardware exists which, when the time sharing switch is on, will do the actual saving and reloading of registers.

The system may now be viewed as having created a virtual processor for each program wishing to run. This virtual Processor has its own set of registers and status bits. Many of these status bits are grouped together for simplicity and called the "flag" register (symbol F). Overflow, extend mode, and two's complement mode status bits are concatenated to the high order end of the fifteen bit PC and called the G register. The state of the hardware processor can thus be stored in five 18-bit words: A, I, X, F and G. To aid system-user program communication, and additional 18-bit register called the W register was added to the virtual Processor. Of course, since this is not a hardware register it can only be accessed through the system software. This

virtual processor is the conceptual "thing" that executes a user's program and is known as a process.

A second problem of time sharing is that, in the general case, it is not desirable to let one user modify the contents of the core memory being used by someone else. To prevent this, the total core memory space is partitioned by the system. Physical core 7 is reserved for the permanently resident parts of the system, while physical cores 0, 1, 2 and 3 may be given to users. The user is then allowed to access only his allotted portion of physical core.

Memory partitioning must be done cleverly, though. It would be highly undesirable to require a program that fits in 4K to fit in a particular 4K, say core 1. If a program is written that references physical core 1 now, but tomorrow the system allocates the program physical core 0, the program couldn't run. Programs must somehow be set up to be able to run in either case. A core-rename register (CR) exists for just this purpose, and can be set only by the system. It allows the system to allocate physical memory space to a user such that when the user program references what it calls core 0 the hardware will automatically redirect the reference to the correct physical core. By suitable partitioning of core, use of secondary storage space, use of the CR register, and use of paging techniques, the system can and does create a virtual memory space for each user. This space may vary in size from as large as six 4K core regions to as small as one.

The third and final difficulty to be mentioned here is that of allocating system resources other than memory and CPU time to the time shared programs. Under a batch processing system or on a dedicated machine all system resources can simply be given to the currently running job. In time sharing mode such a scheme will fail completely. A line printer assigned in such a fashion might print three or four characters for one program, a few from the next scheduled program, etc., resulting in a thoroughly useless printout. It thus becomes desirable for the system to assign resources to individual programs. The program may then use the resource as needed, and when finished with it the program requests the resource be deassigned. The system maintains a list of resources assigned to each computation and this list is called a capability list or C-list. Further discussion of C-lists is postponed until section 5A.5.

A virtual memory space, any virtual processors (processes) that might be executing inside that memory space, plus the list of associated resources (C-list) comprise a sphere of protection. A few examples of spheres are given here, but elaboration on spheres is left for part B of this memo. The time sharing system itself, including the scheduler, I/O controllers, and resource management routines is a sphere. Also, each user is given his own sphere by the system which

is separate from that of other users and in which the user's programs are typically run.

The reader is now prepared for the material which follows.

5A.1

The W register

As stated before, each process has an 18-bit software W register. References to it are much slower than references to a hardware register such as the AC or IO. However, various system calls pass data through this register and it is useful for communicating between spheres via entered processes, as discussed later in section B. It may be referenced by the following instructions.

Instruction	Action
mta 0	Copy A into W
mta 1	Copy I into W
mta 2	Copy W into A
mta 3	Copy W into I

5A.2

Low Priority Mode

The system logically maintains three waiting areas for processes able to run. The first area is where regular priority processes compete for scheduling time. If no processes are in this area, then low priority processes able to run may compete for scheduling time. Finally, if no processes are in either area, a process known as the "hung" process will run. The hung process is the only process ever in the third area.

Occasionally programs are written that would like to run either continuously or not at all; e.g. real time simulation programs. In such cases, use of low priority mode is suggested.

A process may enter low priority mode by executing a mta 100. If this process should execute a frk, the new process will also be in low priority mode. (See part 4 of the Instruction memo for a description of frk.)

5A.3

Real-Time Clock

The real time clock described as device 76 in section 5A.10.5.6 of this memo can also be read by executing a mta 103 instruction. The returned A and I are the same as if hardware I/O device 76 had been assigned and invoked.

5A.4

Virtual Memory Assignment

The memory space available to a user who has just logged in is one 4K memory region. Most programs written will probably fit easily into this space. For those that desire additional space the size of virtual memory may be expanded to as large as six 4K regions or 24K words.

The memory bound of a sphere is equal to one plus the highest numbered address of core owned by the sphere. In the initial memory space available, the highest core address assigned is 7777 octal. Therefore the memory bound is $7777 + 1 = 10000$ octal initially.

Attempts to reference nonexistent virtual memory are illegal. Usually ID will be informed of the error and will print address<< and the error-causing instruction. A method for catching attempts to reference nonexistent memory is described in section 5A.8.

Attachments are discussed later in section 5B.2. In general it may be said that if you do not know what an attachment is, you are not likely to have any.

Instruction Action

mta 206

Read memory bound and attachments into A.
Format is given below.

A	attachments										
	bound					0	1	2	3	4	5
	0	2	3	5	6	11	12				17

mta 207

Set memory bound. The number of cores is specified in A(3-5) or, if that is zero, A(15-17). The memory bound cannot be lowered to zero nor raised above 60000. The content of any existing cores is unchanged. If the memory bound is raised, any attachments that are in the way are automatically removed. Skip if successful. If unsuccessful, the memory bound and all attachments are unchanged.

Illegal Instruction Return and Illegal Memory Reference Return

The illegal returns tell the system that the program has a subroutine of its own to handle the chosen "error" which should be used in place of the "standard" action performed by the system. While both these illegal returns are initially disabled, they may be enabled simply by giving the address of the subroutine to be used.

If an illegal condition occurs and the appropriate illegal return is set then the following will occur:

- 1) The process executing the current instruction will stop in mid-instruction. In most cases this will leave the process and core-memory in the same state as it was in prior to executing the error-causing instruction.
- 2) The process state is modified as follows: W(3-17) will be set to the address of the illegal instruction; W(0-2) is zeroed; G(3-17) is set to the illegal return address specified. Nothing else, including AAL, is modified. This may require the first instruction of the error handler to be a nop or rpf.
- 3) The process will attempt to resume executing code. Be very careful of such lossage as referencing nonexistent memory inside the illegal memory reference subroutine. This may cause a very curious infinite loop.

An illegal memory reference is defined as an attempt to access nonexistent memory. Note carefully that an instruction that attempts to reference locations 00000-00077 inclusive while PRL is on is considered an irrecoverably illegal instruction. Besides, that is memory that exists to the sphere -- it's just protected. The illegal instruction return catches any recoverably illegal instruction. Any illegal system call involving mta or ivk instructions is recoverably illegal. All other types of illegal instructions are intrinsically irrecoverably illegal. (Section 5B.4.1 describes how any illegal instruction may be made to appear recoverably illegal in certain cases.)

One obvious use of these illegal returns is to make a program more self-reliant and less demanding of ID's error printing routines and the programmer's patience. A less obvious use of the illegal memory reference return is to use it to implement a paging routine. The program would be written such that if ever some paged out data were to be referenced, say by having indirected through a dispatch table, then the address referenced would be nonexistent. This would activate the paging routine, which would immediately find the instruction causing the fault and thus the

paged out data item desired. After paging in the data and updating the dispatch table it could then return to the error-causing instruction and resume processing.

Instruction	Action
mta 200	Read illegal instruction return location into A.
mta 201	Set illegal instruction return from A. Any negative number in A will disable this feature.
mta 202	Read illegal memory reference return location into A.
mta 203	Set illegal memory reference return from A. Any negative number in A will disable this feature.

The following program will dismiss normally.

```

100/      iam
          law .+6      / subroutine address for mta 203
          NAX          / makes (X) < 0
          mta 203      / set illegal memory return
          aam          / see below
          lac i        / references the negative address
                   -107
          hlt          / will never be executed
          dsm          / illegal memory reference return
                   / address
A contains 107        / when dsm is executed
X contains -107
W contains 105
G contains 107
F contains 500000    / AAL still on

```

A time sharing system is continually faced with the problems of resource allocation. The problem arises because of the need to prevent interference among concurrently running programs which wish to use a common system resource. There are three kinds of solutions to this problem. The first solution is to provide a "BUSY" or "IN USE" indicator for the resource and then let all programs directly access the resource. Such a primitive system is frequently used for I/O devices, especially by IBM. However, in the case of a disk or drum such a busy flag would likely only indicate when data was being transferred, and some other precautions would have to be taken to insure that no two programs accidentally used the same track or field.

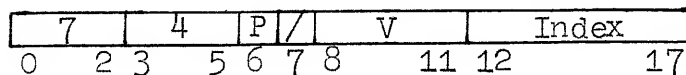
A second solution is to assign to each resource a list of those programs allowed to access the resource. This technique is referred to as an access-list- or list-oriented protection system. Such a system is employed by MULTICS for each segment (file, directory, or whatever) in the system.

The third solution is to assign to each program (more accurately, each sphere) a list of those resources which it can access. This is the approach adopted by the PDP-1, and is referred to as a ticket-oriented system. The system assigns the sphere a resource by handing it a ticket (capability) which will allow admission to system routines controlling that resource or, sometimes, to the resource itself. This capability is not itself the resource but is merely a certification of the user's privilege to use a resource. A program acquires a capability to a resource by asking the system for it. If the system has the necessary resources available to satisfy the requested amount then it will allocate these resources to the user and give the user a capability stating his right to access the resources.

Once the user program acquires a capability there are many things it may do with it. It may ask the system to duplicate the capability. Like a Social Security number, a capability states the program's access rights to a system resource. Perhaps the user program is run by the government and would like to have this privilege presented in triplicate. Another thing that could be done would be to delete the capability. Of course, the system knows whether or not other copies of this capability (records of accessing rights) are still outstanding and so will not free the associated resources until all copies have been destroyed. Many other things may be done to the capability itself as will be discussed in later sections of this memo.

The main reason for acquiring a capability is, naturally, to use the associated resources. This is done by invoking the capability and specifying what action should be per-

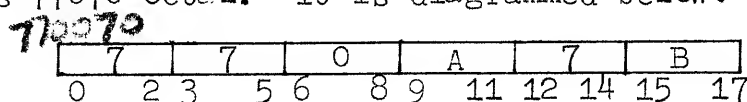
formed. The desired action is conveyed by a process's A, I or W registers, core memory, and/or the ivk (short for invoke) instruction itself. The ivk instruction is diagrammed below.



Bits 12-17 of this instruction are the offset of the capability to be invoked in the C-list. This index is usually known readily to the programmer. The four bits 8-11 labelled "V" are called the variant. Frequently these four bits help specify what operation is desired of the invoked capability. Bit 6 is pause mode and is only important when working directly with hardware devices. It is included here for the sake of completeness since it is never needed by a user in time sharing mode.

As an example, consider an ivk 200 instruction, which most readers should know is used to type in one character to the AC. This instruction has a variant of 2 and a specified index of 0. Usually the user will have a typewriter capability at index 0 of his C-list. This capability has associated with it the physical typewriter which the user would be sitting next to. Variant 2 tells the system what type of operation to perform on this resource -- type in, put code in AC. When the operation is finished the ivk will "complete" and processing will resume.

There exists another class of system calls known as meta instructions. Like the word metaphysics, meta is used here to mean "beyond". Each of the mta (short for meta) instructions is indeed a hardware instruction, but the action it performs depends upon the system software. Formally mta is defined as 77070 octal. It is diagrammed below.



mta AOB

There are 100 octal different mta instruction possible, ~~many~~ ^{some} of which are unused.

As the name implies, these instruction can do more than most other hardware instructions. From the point of view of the system they are subroutine calls, which include requests for the assignment or deassignment of various system resources and for capability duplication or manipulation. From the user's point of view a mta is a single instruction somewhat akin to a microprogram skip instruction (e.g. A+IA=) ---- the instruction performs some specified opera-

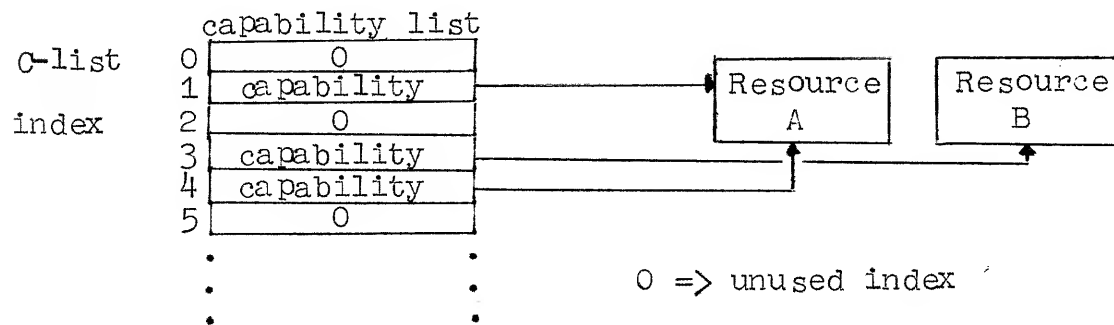
tion using the state of the registers, maybe puts an answer back in one or more of the registers, and possibly skips. The convention that has been adopted for system calls is that a mta instruction will skip if successful IF there was a possibility of failure. Some requests can always be granted.

IN ALL THE FOLLOWING SECTIONS, BOTH IN PART A AND PART B, FOR EVERY SYSTEM CALL OF ANY TYPE, IT IS ASSUMED THAT THE INSTRUCTION CAUSING THE SYSTEM CALL WILL NOT SKIP UNLESS SPECIFICALLY STATED OTHERWISE. This applies to both mta and ivk instructions. In addition, any capability-related instruction, except mta's 402, 403 and 404, will be illegal if the sphere in which the instruction is executed has no C-list.

The mta instructions tend to fall into groups. Mta's numbered less than 204 are non-capability oriented system requests such as read/write W register, read real time clock, read absolute drum field, hlt (mta 4) and dsm (mta 7). Mta's 206 and 207 refer to virtual memory assignment, as already described in the preceding section. Mta's 300 to 307 are requests to assign some type of system resource and grant a capability. Capability and C-list manipulation, including capability duplication and moving, are performed by the mta 400's, except for mta's 406 and 407 which refer to process hoards (see Instruction Memo part 4). Of course, specific numbers mentioned here are subject to change at any time.

The reader should now understand, and should insure that he understands, the difference between a resource and a capability, between deleting a capability and freeing a resource, and between operation on the capability as an object in itself and invoking the capability to use the associated resource. Armed with this knowledge, the reader should have no difficulty with the net sections.

The diagram below illustrates the interrelations among a capability list, a capability, and the system resource the capability refers to. A capability is the system's declaration of a user's right to access some resource. Duplicating a capability at index N in the capability list onto capability index J is like saying "Since I can already access this resource by invoking capability N, let me also access the same resource by invoking a capability at index J." As long as anyone still has a capability to a given resource the resource is not assignable. When all capabilities referencing some resource are deleted, the associated resource is returned to the free pool and becomes assignable by any sphere.



The following is a list of system calls useful for general manipulation of capabilities. The functions include duplication, deletion, exchanging, and counting. Mta 400 allows the program to discern whether or not a given capability index is used, and if so, what kind of capability is at the capability index.

Instruction	Action
-------------	--------

mta 204

Delete capability. A(12-17) specifies the index of the capability to be removed. If no capability exists at the given index, no action occurs. A, I unmodified. Does not skip.

mta 400

Read capability at index specified by A(12-17). The 18-bit word describing the capability is put into A. If no capability exists at that index, A is cleared. Does not skip.

mta 401

Exchange capabilities. The capabilities at the indices in A(6-11) and A(12-17) are swapped. There need not exist capabilities at the specified indices. Does not skip.

mta 404

Count capabilities. Return in A the total number of used capability indices. Returns 777777 if no C-list exists.

mta 405

Copy capability. The capability at the index in I(6-11) is copied into the index in I(12-17) (or the first free index if I(12-17) is zero). If successful, skip returning the index used in A and a copy of the capability in I. If unsuccessful then

- 1) if a capability already exists at the index in I(12-17) (or at all indices if I(12-17)=0), then a copy of the interfering capability (or the last capability) is placed in I. A will be unchanged.
- or 2) if either no capability or an entered process capability exists at the index I(6-11), I will be cleared and A will contain I(12-17) (or the first free index if I(12-17)=0).

As discussed in sections 5A.0 and 5A.7, a sphere includes a capability list to allow processes to access the outside world. There are two possible places to put the list: inside the depths of the system or within the user's own memory space. If it should be elected to put this list inside the system, and if it is desired to keep the memory space used by the system within the limits of the physical hardware, then there must be a definite upper limit to the total space available for storing C-lists.

On the PDP-1, both modes are used. The system will keep for each initial user sphere a C-list allowing indices from 0 to 17 octal. All other spheres, including all spheres that user programs can create, must explicitly allocate space inside their own memory space for the storing of a C-list, if a C-list is desired. This is accomplished by executing a mta 403 to enter PRL (program reference list) mode. Locations 0-77 of the core 0 of the sphere will then be used for storing capabilities.

Special hardware exists in the machine to keep the user program from examining or modifying these reserved locations when PRL is on. Any attempt to read from or write into these locations will produce an irrecoverable illegal instruction fault. A user expecting to run with PRL on should therefore be careful to assemble his programs beginning at or after location 100.

Use of special hardware devices not reserved exclusively for the system requires that the user be in PRL mode. This is the case for the teletype devices 16 and 17, the Calcomp plotter and others listed in section 5A.10.5.6, Hardware I/O Devices.

Unfortunately, the system routines that recover after a system crash will only recover capabilities on indices from 0 through 17 (octal). The effect is the same as if PRL had been off at the time of the crash.

The cks instruction (720033) may be used to determine whether or not PRL is on. If bit 8 of the word read into I by the cks instruction is a one, PRL is on. Otherwise PRL is off. Other bits ~~historically perform useful functions~~, but are now obsolete.

↑
that used to be useful

Instruction Action

mta 403

Turn on PRL. Core 0 locations 0 through 77 octal become protected. Bit 8 of the cks word will be turned on.

mta 402

Turn off PRL. Core 0 locations 0 through 77 become unprotected, and are set to zero if PRL was on at the time of executing this instruction.

Note: For the initial user sphere only, when PRL is turned on current capabilities in indices 0-17 are copied into the PRL C-list in the corresponding indices. When PRL is turned off, capabilities with indices from 0 to 17 inclusive are saved, and those from 20 to 77 are deleted. Spheres other than initial user spheres will have all their capabilities deleted if PRL is turned off.

There are several items which may be assigned to a sphere but which will not be expressly entered into the C-list. These items are assigned and deassigned by using a mta 500 instruction, with the appropriate code in the AC. The external levels are also listed for convenience.

A(0-5) Code	Action
0	Assign external levels. More than one may be assigned in a single instruction. For all i , $1 \leq i \leq 7$, external level i is assigned if $A(i + 10) = 1$. Skip if ok.
1	Deassign external levels as above. No skip.
2	Assign external register, common. Skip if ok. If one sphere has the ext. reg. owned in common mode then any other sphere requesting the ext. reg. in common mode will have its request granted. Any sphere owning the register in this mode may read and write the contents of the register at will.
3	Assign external register, private. Skip if ok. No spheres may use the register except the sphere to which the register is assigned. This is the desired mode for music players, although <i>tsmplay</i> actually assigns it in common mode.
4	Deassign external register. No skip.

External Levels

- 3 Used for temporary hardware connections
- 4 External clock
- 6 Used for temporary hardware connections
- 7 Radio Astronomy antenna

The group of instructions from mta 300 to 307 is used to acquire capabilities to system resources. There are eight categories of resources: 4K drum fields, entered processes, spheres, program queues, directories, files, hardware I/O devices and entries. The category of entries is used to cover those I/O resources which have a software system-resident controller. Examples are the paper tape reader and punch, microtapes and typewriters. The eight categories given are numbered type 0 to 7 respectively.

To a good approximation a mta 30X instruction is used to assign a resource of category X. For example, a mta 305 would be used to assign a file, resource type 5. There are two exceptions. Mta 301 does not exist, and mta 306 is used to assign all I/O devices --- not only hardware devices (type 6) but also all of those with a system controller (accessed through capabilities of type 7).

All of the mta 300 series of instructions use the same general format. The program must specify what resource is desired and where to put the new capability that will be created. Where to put the capability is always specified by A(12-17), and the convention has been adopted that if these six bits are zero then the first free (unused) C-list index will be used, beginning with index 0. This first-free-if-0-is-specified convention is frequently used throughout the system.

All of the mta 300's skip if successful. Success is being able to allocate the requested resource and being able to put a capability at the requested index. If the instruction skips, A will contain the C-list index and I will contain a copy of the capability at that index. If the instruction fails to skip:

- 1) If I is nonzero, then there is already a capability at the requested index (or all indices if zero was requested). A will be unchanged.
- 2) If I is zero, then there are insufficient resources to satisfy the request. A will contain the requested index (or first free index if zero was specified).

Instruction	Action
mta 300	Assign drum field. If A(0) is 0 any available drum field will be assigned and both read and write operations may be performed. If A(0) is 1 then the absolute drum field given by A(5-11) will be assigned in read-only mode. Read-only indicates that write operations are recoverably illegal. The initial contents of a writable drum field are not predictable.
mta 303	Assign program queue. The initial population of the queue is ones complement - I .
mta 304	Assign directory. A directory is initially devoid of capabilities.
mta 305	Assign file. Initial length is zero.
mta 306	Assign device. The particular device class is given by A(0-5) and variations within a class are specified by A(6-11). The major classes are:

- 0 microtapes. Drive number in A(6-11).
- 1 buttons. A(6-11) indicates panel 0 or 1.
- 2 temporary clock
- 3 paper tape reader. Alpha mode chosen if A(11) is zero, binary mode otherwise.
- 4 paper tape punch
- 5 typewriter. A(6-11) indicates desired console.
- 6 call button. A(6-11) indicates desired console.
- 7 hardware device. A(6-11) indicates which device. Devices 1, 2, 20, 21 and 77 are permanently owned by the system.

Each of these resources is described in the following sections along with the ivk instructions that apply.

```

Examples.    law 216      / assign tape drive 2 to
              mta 306     / capability index 16
              hlt         / if unsuccessful
A contains 16 / if successful
I contains <microtape capability>

```

```

              lac (30010   / assign alpha reader to
              mta 306     / capability index 10
              hlt         / if unsuccessful
A contains 10 / if successful
I contains <reader capability>

```

Drum fields are assigned as indicated in section 5A.10.0. The format of data transfers is indicated in Instruction Memo part 3.

The primary use of drum fields is for storing core image data, such as the binary version of a program as produced by the assembler. Read-only drum field capabilities allow a user to access certain system programs such as E.T., Certainly, ET Window and ID. Storage of non-core-image type data such as text should use files in preference to drum fields as files are inherently more efficient use of system resources.

Mta 104 and mta 105 are also used to read any absolute drum field. The format is the same as for a drum ivk, except that the drum field comes from I(0-5). Mta 104 reads fields 0-77, mta 105 reads fields 100-177. Mta 104 and mta 105 skip if no error occurs. The registers are always unchanged.

Since program queues are discussed thoroughly and competently in part 4 of the Instruction Memo, only brief comments are given here. Queues are described solely for the sake of completeness.

A queue provides a way for a process to ask the system for conditional descheduling. The process may remain descheduled for an indefinitely long period of time. This and other features make a queue an efficient lock mechanism, preventing more than some desired maximum number of processes from entering the locked region of code.

All registers are always preserved by queue ivks.

Variant	Action
0	A(13-14)+1 is used as the variant
1	Enter queue. The population is increased by one. If the result is strictly positive, the process is suspended in the queue. If the result is zero or negative, the instruction completes and the process continues.
2	Release queue. The population is decreased by one. If the result is nonnegative the process that has been suspended longest is removed from the queue and restarted as if its enter queue had just completed. The release queue ivk always completes immediately.
3	Read queue population into A. Primarily used for debugging.

A directory is a means of storing capabilities. Each directory can hold as many as 100 (octal) capabilities. A capability in a directory, however, may not be directly invoked but must first be placed in the user's C-list.

A directory may contain any kind of capability. There are two basic kinds of directory capabilities. The first is the kind created by the create directory mta. This capability may be invoked to add things to, delete things from, or otherwise modify the directory. The second variety is a read-only capability. This kind of capability may only be invoked to obtain a capability from the directory. Attempts to modify the directory using this capability are illegal, except retrieval of entered process capabilities which return error 106.

Directories are assigned as indicated in section 5A.10.0.

The following ivks exist.

Variant Action

0

A(8-11) + 1 is used as the variant.

1

Retrieve capability. The capability specified by I(6-11) in the directory is placed at I(12-17) in the user's C-list. The ivk will skip if a nonzero capability was successfully placed at the desired index in the user's C-list. The capability in the directory is not deleted unless the capability was an entered process capability. If I(12-17) is a zero, the first free index in the user's C-list will be used. The contents returned in A will be the index in the user's C-list at which the capability was (or would have been) placed. I will contain a copy of the capability at the index in A.

2

Place capability. Similar to variant 1 above. I(6-11) specifies the index in the user's C-list and I(12-17) the directory index (or if I(12-17) is zero the first free directory index will be used.). The user's copy will be deleted only if the capability was an entered process capability. This ivk will skip if a nonzero capability was successfully transferred.

3

Remove capability. The capability at the directory index given in A(12-17) will be deleted. All registers are left unchanged by this ivk.

7

Count capabilities. Return the total number of capabilities in the directory in A.

14

Convert to read-only capability. The capability invoked is turned into a read-only capability. Only variants 1, 7, 14 and 15 (or appropriate variations of variant 0) are legal on such capabilities. All other variants imply a directory modification. Note, however, that an entered process capability may not be retrieved via a read-only directory capability as this would necessitate a directory modification. In this special case error code 106 (octal) would be returned in A.

15

Translate capability. Similar to a mta 400 on a capability in a C-list. The capability at the directory index specified by A(12-17) is read and placed in A.

Variants 1 and 2 and appropriate variations of variant 0 will skip if successful. Variants 3, 7, 14, 15, and appropriate variations of variant 0 will never skip.

A file is a variable quantity of drum space which may be as little as zero or as much as 1,000,000 (octal) words. The use of files for secondary data storage instead of drum fields is urged since files inherently use space more efficiently. For text, especially, the use of files eliminates the problems associated with use of drum fields where even a short text uses 4K of drum space, and a long text requires several capabilities and the use of computed ivks.

Files are assigned as indicated in section 5A.10.0.

File operations fall into two classes: data transfer between core and the file, and operations on the file as an object, such as read length. The type is determined by A(14). The description that follows implicitly includes this fact.

File ivks are illegal if the variant is illegal, if an attempt is made to write or set length using a read-only capability or a read/write operation specifies an illegal core address.

Read/Write operations.

A file is from 0 to 2000 (octal) blocks of 400 (octal) words each. Data transfers must be entirely contained within a single file block. Thus the maximum transfer is 400 words per ivk, and if the transfer begins at word 340 of a file block, only 40 words can be transferred.

old!
see p.30a

Data transfers must also be entirely within a single 4K core module. Thus a 400 word transfer beginning at 07400 is allowed but 07401 to 07777 is not allowed as the initial address for a 400 word transfer.

no longer true

Writing past the end of file will normally cause the file to be extended sufficiently to create the specified block. Thus writing data into block 50 of a previously zero length file will cause the file to become 51 blocks long (blocks 0 through 50). The contents of the first 50 blocks ~~would be~~ ^{are} unpredictable.

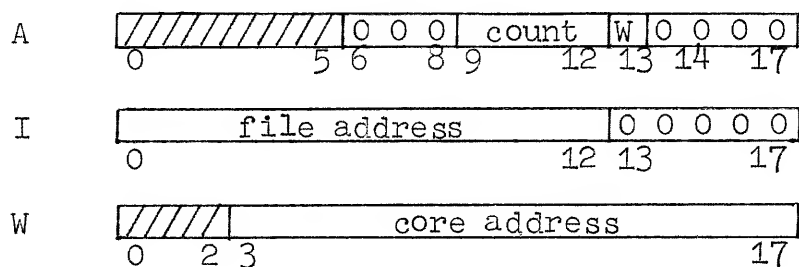
The read/write ivk will skip if successful. If unsuccessful then it will be because:

1. An attempt was made to read a non-existent block of the file.
2. The specified transfer required the crossing of a file block boundary or a core boundary.
3. An attempt was made to write past the end of file and the

file could not be extended far enough to create the specified block.

The format of the A, I and W registers for a read/write operation is similar to that of drum fields and microtapes. A(13), the 20-bit, if zero will read data from the file into core; if one, data is written from core into the file. The count of the number of words to be transferred divided by 40 (octal) is given in A(6-12). A(14-17) should be zero.

The address in the file at which the data transfer is to begin is specified in I. Since this address must be a multiple of 40 words, I(13-17) will always be zero. W(3-17) gives the extended core address at which the data transfer is to begin, and refers to the core of the sphere executing the file ivk. W(0-2) is ignored.



NEW IMPLEMENTATION

Any file ivk that requests a transfer entirely within a single 400-word block will continue to behave as before. When a transfer must cross 400-word boundaries the I and W registers will count up by blocks and the A will count down (much like microtape ivks). At some point the remaining word count will lie entirely within a single block, and when that final transfer has completed, that count along with the associated core and file addresses will remain in the registers. If the ivk is not successful, it may have actually finished part of the requested transfer, and the values in the A, I, and W will point to whatever part of the transfer remains undone. Transfers may actually cross core boundaries ~~as long as~~ ^{provided that} no single file block ever overlaps a core boundary. It should be noted that the invoking process must be restarted after each transfer of a block or fraction of a block, and so a multi-block request that writes over the instruction invoking the file may cause the request to terminate before completion.

For example, say the A, I, and W contain 740, 300, and 530 initially. Then the transfer will take place in three parts as follows:

```
file { 300- 377 } → core { 530- 627 }
file { 400- 777 } → core { 630-1227 }
file {1000-1237 } → core {1230-1467 }
```

When the transfer has completed, the A, I, and W will contain 260, 1000, and 1230, respectively.

~~(Cited on sys.g as "multiblok file document")~~

SEE instr5a 760218

File-Object Operations

These ivks treat the file as a whole entity. If variant 0 is used in the file ivk then A(10-13)+1 is used as the variant. It is suggested that A(15-17) be set equal to 5. These suggested values for use with variant 0 are listed in parentheses after the variant. If the variant is used, the A value is ignored.

Variant (A) Action

- 1 (15)
Read length. The length in 400-word blocks is put into the AC. The result will not be less than zero, nor greater than 2000. Does not skip.
- 2 (35)
Set length. The length in 400-word blocks is set from the I register. Skips if successful. If unsuccessful, then I was set greater than 2000, or there was insufficient drum space.
- 3 (55)
Convert to read-only capability. Write and set length operations on the file via this capability become illegal. Read and read length will remain legal. Does not skip.

should have operations to insert or
delete blocks from the middle of a file.

Also, how about a mta instruction that
magically changes a read/write drum field
to a file if no other copies exist.

Also, how about a file object ivk that
causes some other specified file to be
emptied on top of the first one

- 4 (75) Append contents of file on I(6-11), skip if OK. Returns plundered capability in I. Both must be read/write, distinct files. Second file is emptied
- 5 (115)
Insert a block before block n, where n is in I(0-17)
skip if successful. Block might not have existed & still succeed.
- 6 (135)
Delete block n. No skip

→ Both 5 & 6 cause all blocks $\geq n$ to be renumbered. Beware!

This section is designed to complement Instruction Memo part 3 which describes the operation of I/O devices. Each of the following sections contains any additional use or peculiarity of the various devices that was not mentioned in part 3.

Assignment of I/O devices using mta 306

The mta 306 instruction assigns most of the commonly used I/O devices. As with all mta's, much of the data is passed in the AC. The format referenced here is

high six bits A(0-5) is called CODE

middle six bits A(6-11) is called ARG2

The low six bits A(12-17), called INDEX, should be set to whatever C-list index you wish the resultant capability to be placed at. If INDEX contains 0, the first free capability index will be used. All mta 306 requests will skip if successful. If unsuccessful, A(0-11) will be cleared.

CODE	Action
0	Assigns a microtape. ARG2 should be set to the number of the desired tape drive.
1	Buttons. ARG2 should be the number of the desired button console (0 or 1).
2	Temporary clock. ARG2 ignored.
3	Paper tape reader. If reader should read in Alpha mode use ARG2 = 0. Otherwise use ARG2 = 1 to read tape in binary mode.
4	Paper tape punch. ARG2 ignored.
5	Typewriter. ARG2 should be set to the desired console number. Will succeed if the console is not logged in and if the typewriter is not otherwise owned.
6	Call button. ARG2 should be set to the desired console number. Will succeed if the console exists. Consoles 2, 3, 4, 5 and 6 exist.

Hardware device. ARG2 specifies the desired device number. Devices 1,2,20,21, and 77 are permanently owned by the system.

```

Examples.      lac (216           /assign microtape drive 2 to
                mta 306           /capability 16 octal
                hlt                / if problems
                jmp .              / if all ok

                lac (30010         / assign alpha reader to
                mta 306           / capability 10 octal
                dsm                /if problems
                jmp .-3

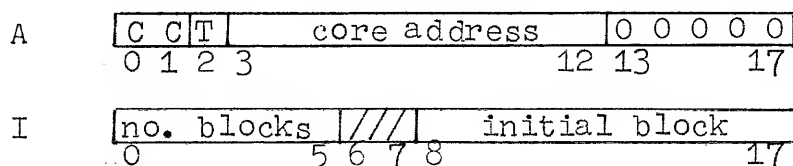
```

Note that in this last example, the mta 306 will skip once at most. This is because 1) the capability is already used the second time around, and 2) the reader has already been assigned to someone and only one copy is allowed to be given out at a time (although there may be many copies of the original capability).

Microtapes are assigned as described in section 5A.10.5. The variant of a microtape ivk is ignored.

Some years ago microtapes were formatted with only 1000 blocks, instead of the standard 1102. While all public tapes and most private tapes have been switched to the standard 1102 format, the system still understands the 1000 block format for compatibility.

Data transfers use only A and I, and their format is shown in the diagram.



I(0-5) contains the number of blocks to be transferred; if zero, one block will be transferred. I(6-7) is ignored. I(8-17) contains the number of the first tape block to be transferred.

A(3-17) specifies the core address of the first word being transferred, and since it must be a multiple of 40 (octal) words, A(13-17) will contain zeros. The fact of A(15-17) being zero is used by the system to determine that 1000-block translation is being requested. A(0-1) are the CC bits and correspond to A(13-14) in 1102-block format tape ivks. A(2) is the translation, or T, bit and corresponds to A(12) in 1102-block format. In the normal case where translation is on (T bit is off) the block number is interpreted mod 1000 (octal).

Restrictions about transfers crossing core boundaries are the same for all tape ivks.

If all blocks are transferred successfully, the ivk instruction skips, leaving the address of the last word transferred +1 in A(3-17) with A(0-2) unchanged, and the number of the last block transferred +1 in I. If an error occurs on any block, the instruction does not skip, I(8-17) contains the number of the block in which the error occurred, and I(0-5) contains the number of blocks remaining, including the one that was in error. In addition, the standard error code will be returned in the W register.

In 1000-block mode, the system translates the blocks so that logical tape blocks 0, 1, ..., 377, 400, 401, ..., 777 reference physical tape blocks 1, 3, ..., 777, 776, 774, ..., 0.

There are two button consoles available, numbered zero and one. More detail may be found in Instruction Manual part 3. Button capabilities are assigned as indicated in section 5A.10.5. *Note that the existing panels may always be assigned successfully.*

Invoking a button console capability will cause the process executing the ivk to hang until the state of the buttons differs from the contents of the AC. When that condition is met, the process will be restarted and the contents of the AC will be the current state of the buttons in the same format as read by a rbt instruction. The rbt instruction works independently of button console capabilities.

The system provides a clock which ticks 1760 times per minute, or slightly less than 30 decimal ticks per second. The clock is assigned as indicated in section 5A.10.5, and an very large number of clock capabilities are available.

unlimited
A process invoking a clock capability will hang as follows:

- 1) if the ^{value in}~~contents~~ of A is positive, plus or minus zero, processing will resume immediately and the contents of A will be unchanged;
- 2) if the ^{value in}~~contents~~ of A is less than 777777, each clock tick will (one's complement) add 1 to the contents of A. When the contents become zero the process will be unhung.

All arithmetic on clock time is done in one's complement. If time larger than 1 hr., 14 min., 28 and 29/88 seconds is needed, the user should write a loop into his program that invokes the clock as many times as necessary to get the desired total waiting time.

The following programs are offered as examples.

1. law i 30. / will wait slightly over a second
 ivk 10 / if a clock capability is at index 10
 sza
 hlt / will never be executed
2. law 30. / will not wait at all
 ivk 10 / (clock on capability 10)

5A.10.5.4 Paper Tape Reader and Paper Tape Punch

The operation of these devices is described in Instruction Memo part 3.

Only one process at a time may operate the devices. Additional processes that try to invoke the device will get a function busy error.

The initial user sphere is provided a typewriter capability at index 0. If a console is not logged in and no one else already has a capability to the associated typewriter, then that typewriter may be assigned by using a mta as described in section 5A.10.5.

Variants 0 through 4 and 13 through 16 are described in Instruction Memo part 3. The next section is best understood only after reading part 5B of this memo about spheres.

It is occasionally desirable to share a typewriter with another program, frequently an inferior sphere, but in such a way that the original owner may assert control over the typewriter when it wishes, without finding and deleting all copies in the recipient. For example, ID gives the initial user sphere a typewriter at capability index 0. However, when the user hits the call button, ID wants control of the typewriter, and would not appreciate user programs trying to type in or out. This facility is provided by having inferior typewriter capabilities. The recipient of an inferior typewriter capability is not aware of the enabling and disabling operations. Inferior typewriter capabilities may be created to any reasonable depth.

Creating an inferior typewriter capability involves the following steps:

- 1) Given a typewriter capability at index k, duplicate the capability onto index j.
- 2) Convert this new typewriter capability at index j to an inferior capability by using the ivk given below. This is now an inferior typewriter capability with enable/disable permit.
- 3) Duplicate this capability still at index j onto capability n.
- 4) Convert this into an inferior typewriter capability without an enable/disable permit using the ivk listed below.
- 5) Give this new capability at index n to whoever needs it.

Enable/disable permit is used to suppress both typein and typeout on all inferior typewriter capabilities. When a process invokes a disabled typewriter capability, it will be

hung until the capability is enabled. Inferior typewriter capabilities with enable/disable permit may be ivk'd for typein or typeout if desired.

Typical use of an inferior typewriter capability is illustrated by ID. ID has two typewriter capabilities; the one it has at index 0 of its C-list corresponds to the initial typewriter at index k above, and the one it has at index 12 corresponds to the inferior capability at index j above. When the user hits call or otherwise causes ID to run, ID suppresses the inferior typewriter at index 12 before typing out on its typewriter at index 0. When ID has finished running, it will re-enable the typewriter at index 12, allowing the user's initial typewriter and any inferiors not otherwise disabled to resume operation.

Variant	Action
5	Enable.
6	Disable.
7	Unused.
10	Unused.
11	Turn off enable/disable permit for ivk'd capability.
12	Convert to inferior typewriter. The typewriter capability ivk'd is replaced by an inferior. Skip if successful. The new capability will be disabled and will have an enable/disable permit. A copy of the capability is placed in I.

The design of the PDP-1 allows some I/O operations to be performed directly by users. An extensive and somewhat accurate description of this facility may be found in memo PDP-33, Input/Output in the PDP-1X. PRL must be on to operate hardware directly (see section 5A.8).

Assignment of hardware devices is described in section 5A.10.5. The following device numbers are currently used:

1	New drum side A
2	New drum side B
16	Teletype input
17	Teletype output
20	Microtape unit monitor
21	Microtape data controller
25	PDP-11 link transmitter
26	PDP-11 link receiver
27	Calcomp plotter (see below)
30	Clock (see below)
31	Real-time clock alarm device (see below)
32	Special user device.
76	Real-time clock (see below)
77	Microtape motion controller

Some of these devices are described in separate memos. Most require special turn-on procedures. Devices 1, 2, 20, 21, and 77 cannot be assigned.

Calcomp Plotter

The use of the plotter is fully described in Instruction Memo part 3. The plotter is assigned as noted in section 5A.10.5. The variant of the ivk is ignored.

Clock

This is an adjustable speed clock designed to tick in the vicinity of 60 times per second. The clock is assigned as device 30 as described in section 5A.10.5. All ivks on this clock will hang until the clock "ticks". All registers are unchanged by this ivk.

Ticking rates of greater than about 365 times per second are not possible because of the system scheduling overhead needed. The minimum rate of ticking is around 2.1 ticks per second.

The tick rate is adjustable with knobs in bay 10.

Programs wishing to keep track of long time periods by using this device might employ one of the techniques exemplified below.

```
ivk 66      / device 30 assigned to capability 66, PRL on
idx time    / will cycle to zero in 1 hr, 12 min,
jmp .-2     / 49 1/15 sec assuming 60 ticks per second

ivk 66      / same as above
idx time    / will cycle back to zero in 6 years, 17 days,
sza         /      8 hours, 17 minutes, 15 17/45 seconds
jmp .-3     / (assuming two leap years) given a tick rate
idx time2   /      of 360 ticks per second
jmp .-5
```

Real Time Clock

The real-time clock is implemented as two devices, the clock device (hardware device 76) and the alarm device (device 31).

The clock device maintains a 36 bit time register, which is incremented every 100 microseconds. It may be read but not written. The time register overflows only about every 79 days.

The alarm device is usable as a timer for intervals of up to 26.2144 seconds. It maintains an internal 18 bit register which counts down by one every 100 microseconds, and which turns on a flag when it reaches zero.

For the alarm device, variant 17 acts as an I/O clear. The execution of any nonwaiting variant is legal even if another process is at that time waiting on the alarm, and it will have its normal function. Attempting to execute a waiting variant while another process is waiting causes a function busy error. Variants 0 and 1 are the only waiting variants.

Device	Variant	Function
76 (clock)	any	Read current contents of time register. A contains the high 18 bits. I contains the low 18 bits.
31 (alarm)	0	Clear flag, load alarm register from I, and wait. When flag comes on, clear it again and complete.
	1	Wait. When flag is on, clear it and complete.
	2	Clear flag, load alarm register from I, and complete immediately.
	3	Test flag. Skip if flag is on. Clear flag and complete immediately.
	17	I/O clear. Clear flag and cause any other process now in an alarm wait to complete. Complete immediately.

/ This section of the memo is generally unstructured. It does not abide by the rule that only knowledge from previous sections is needed to understand everything in this section, though this might be true by chance. Hopefully this section of the memo will be of value to users.

Short Glossary


administrative routine (AR)

A part of the time sharing system. It is not permanently locked in core but is paged in as needed. All mta's ≥ 200 are processed by the AR as are many ivk instructions. It is responsible for keeping track of all resources except CPU time and for capability manipulation. The microtape controller resides in the AR. The AR usually runs in user mode.

capability

An 18-bit word stating a privilege to use some associated resource.

C-list

Capability list. A list of either 20 or 100 (octal) elements, some of which may contain capabilities; associated with a single sphere. 

executive routine (ER)

The part of the time sharing system that is permanently resident in core 7. It contains the scheduler, paging routines, and most I/O controlling routines. It performs some of the simple operations on files. Many ivks and all mta's < 200 are performed by the ER. The description of every sphere, process, and non-PRL C-list is kept there.

first free convention

See section 5A.10.0, paragraph 3.

initial user sphere

Currently, when a user logs in, the system creates two spheres. One is the user's ID and the other is the sphere owned by ID in which most user programs are run. This second sphere is the initial user sphere.

invoke

(Webster) to call on for assistance or protection;

to demand judicially

memory bound

One greater than the highest address of core owned by the sphere. Note that attachments are not considered owned core.

meta-

(Webster) A prefix used in words of Greek origin to mean in the midst of, among, between, beyond, after, reversely.

Process

The conceptual "thing" that executes code in a sphere. A virtual processor consisting solely of its A, I, X, F, G, and W registers.

scheduler

Part of the executive routine responsible for equitable distribution of CPU time to processes. Occasionally blamed by irate users.

sphere or sphere of protection

A virtual memory space, any virtual processors executing in that memory space, and perhaps a C-list compose a sphere. A sphere is roughly equivalent to a complete virtual computer whose I/O devices are the capabilities of a C-list.

Comments

Some programming conventions have developed which are used frequently by the system and system hackers. These conventions have developed through convenience and personal taste and in no way are necessary restrictions of the system.

Capabilities 0-17 tend to be used roughly as follows. That is, given the index, what is most likely to be found there.

- 0 typewriter; mostly because ID puts it there
- 1 binary copy of program. Observed by assembler and file system.
- 2-6 ET text, hence drum fields
- 7 beginning of alternate ET buffer. Hence frequently used for text generating programs like the justifier.
- 10 reader. Mostly because ET uses it for that
- 11 no tendencies are known. Might be anything
- 12 sphere, queue or inferior typewriter
- 13 usually a queue
- 14 almost always a sphere

- 15 drum field. Usually a utility program binary
of some sort.
- 16 scratch. Many system programs when starting up
↑ a file system will automatically delete anything
on index 16 before putting the microtape there.
Programs include ID, ET, and the File System.
- 17 a place to hide a capability; usually considered
a "safe" index since practically nothing ever
uses this index.

The punch, an extra typewriter, button panels and the like tend to be assigned to any index or to first free.

Programs written to last into the future should use files in preference to drum fields. Access time is about the same but files are more efficient use of space.

A program should clean up after itself. Don't leave lots of stray capabilities lying about in the C-list.

If a capability-generating instruction, e.g. mta 405 or mta 304, fails to skip, section 5A.10.0 suggests an interpretation of the values of A and I returned.

Use the mta summary --- it's faster than looking it up in the memo. It's also cross referenced in case additional information is needed.